SSP: Eliminating Redundant Writes in Failure-Atomic NVRAMs via Shadow Sub-Paging

Yuanjiang Ni
UC Santa Cruz

Jishen Zhao
UC San Diego

Heiner Litz
UC Santa Cruz

Daniel Bittman
UC Santa Cruz

Ethan L. Miller
UC Santa Cruz
Pure Storage

ABSTRACT

Non-Volatile Random Access Memory (NVRAM) technologies are closing the performance gap between traditional storage and memory. However, the integrity of persistent data structures after an unclean shutdown remains a major concern. Logging is commonly used to ensure consistency of NVRAM systems, but it imposes significant performance overhead and causes additional wear out by writing extra data into NVRAM. Our goal is to eliminate the extra writes that are needed to achieve consistency. SSP (i) exploits a novel cache-line-level remapping mechanism to eliminate redundant data copies in NVRAM, (ii) minimizes the storage overheads using page consolidation and (iii) removes failure-atomicity overheads from the critical path, significantly improving the performance of NVRAM systems. Our evaluation results demonstrate that SSP reduces overall write traffic by up to 1.8×, reduces extra NVRAM writes in the critical path by up to 10× and improves transaction throughput by up to 1.6×, compared to a state-of-the-art logging design.

CCS CONCEPTS

• **Information systems** → *Phase change memory*; • **Computer systems organization** → *Reliability*.

KEYWORDS

shadow sub-paging, NVRAM, failure-atomicity

ACM Reference Format:
Yuanjiang Ni, Jishen Zhao, Heiner Litz, Daniel Bittman, and Ethan L. Miller. 2019. SSP: Eliminating Redundant Writes in Failure-Atomic NVRAMs via Shadow Sub-Paging. In *Proceedings of The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3352460.3358326

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO'19, October 12-16, 2019, Columbus, OH, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358326

1 INTRODUCTION

Non-Volatile Random Access Memory (NVRAM) is becoming a reality, as technologies such as STT-RAM [20], PCM [39] (Phase-Change Memory) and memristors [45] show DRAM-like performance and disk-like persistence. Recently, Intel has announced its Optane DC persistent memory [17], further facilitating the acceptance of NVRAM as a new storage tier. NVRAM on the memory bus ("persistent memory") introduces a new storage interface: applications use CPU load/store instructions to directly operate on the storage medium. This model removes the need to maintain separate data formats for memory and storage. The resulting programs are streamlined and reduce overheads, for instance, by avoiding serialization and deserialization of the in-memory data structures.

Since data in persistent memory survives a power cycle, this model requires mechanisms to ensure that persisted data is reusable by preserving application consistency in the presence of failures. Durable transactions provide a straightforward abstraction to support consistency—programmers only need to specify the code that should be part of a failure-atomic section. The updates within the failure-atomic section are guaranteed to be executed indivisibly (all or nothing).

Prior work on providing failure-atomicity for persistent memory [4, 5, 14, 18, 19, 44, 49] has focused on the following challenges: First, the limited write endurance [23] of NVRAM compared to DRAM, second, the latency overheads introduced for guaranteeing failure-atomicity, by memory fences, flushes and logging and third, the inability to amortize overheads when operating on byte addressable NVRAM. Prior work has approached the above challenges with three techniques: write-ahead logging, log-structuring and shadow paging.

Write-ahead logging [4, 9, 16, 49] ensures storage consistency with explicit data copying. Several previous works [4, 9, 16, 49] employ fine-grained logging to avoid unnecessary memory bandwidth consumption. Unfortunately, logging, even at finest granularity, still causes redundant writes that introduce bandwidth overheads [54] and cause additional wear out. Log-structured stores only maintain a single copy of a data element, but they must adjust a mapping table on each write that references the new element. Furthermore, the log-structured approach usually maintains mappings to data elements of large size to reduce the capacity overhead of the mapping table in comparison to the actual data. Unfortunately, this introduces fragmentation, requires large page writes and introduces garbage collection overheads, similar to those incurred by NAND Flash [24]. It is required to use a more flexible, and hence complex, mapping scheme [14] to reduce the otherwise prohibitive metadata

overhead for persistent memory systems. The third approach that has been studied relies on shadow paging and its optimizations [5] leveraging Copy-on-Write (CoW) [13]. In CoW, modified data is only written once, however unmodified data has to be copied first, rendering this approach inefficient for small updates.

In this work, we introduce a hardware-friendly remapping mechanism, based on shadow paging, that can i) reduce the number of extra NVRAM writes over logging, and ii) avoid extra data copying within the critical path. We propose Shadow Sub-Paging (SSP), that supports cache-line-level remapping with low metadata overhead: our approach requires only three bits for each cache line in pages that are being actively updated. The process of atomic updates and transaction commit only involves updating the per-page metadata using simple bitwise operations and no extra data movement is required in the critical path. We extend the translation lookaside buffer (TLB) hardware to support SSP semantics, minimizing changes to the processor core while avoiding most address remapping overheads.

SSP leverages the following key observation to provide failure-atomicity: Instead of duplicating writes as in logging based approaches, SSP only consistently update a small amount of meta data in NVRAM. As the metadata is small compared to the actual data, redundant write traffic, which is a major concern in existing NVRAM systems, is almost completely avoided. However, SSP raises two new challenges. First, metadata needs to be written in an atomically consistent way, which we address with *lightweight metadata journaling*. And second, SSP introduces memory capacity overheads, which we address via *page consolidation*.

In a nutshell, our approach works as follows: For each virtual NVRAM memory page in the TLB, SSP maintains two physical page mappings. Persistent writes are applied to the two pages alternatively and SSP switches the page mapping on each failure-atomic transaction. Instead of performing CoW on a per page granularity, SSP maintains additional meta information that tracks state on a cache line basis within each page. Finally, when a page is evicted from the TLB, SSP performs page consolidation to merge the two physical pages into one. Page consolidation is the only point where SSP introduces redundant writes. Our key observation, however, is that the number of transactions is much higher than the number of TLB evictions for most applications. This allows SSP to batch redundant writes to NVRAM resulting in a significant decrease of overall writes. As page consolidation can be performed in the background, SSP removes overheads required for failure-atomicity from the performance critical path.

In summary, this paper makes the following contributions:

- We propose an efficient remapping technique which avoids the inefficiency of shadow paging by allowing it to track modifications at cache line granularity.
- We propose metadata journaling to preserve consistency.
- We propose page consolidation to save storage cost.
- We evaluate SSP and show that it can improve performance by 64% for micro-benchmarks and by up to 35% for real workloads over a state-of-the-art logging design.

The remainder of the paper is organized as follows. Section 2 provides background on persistent memory systems, durable transactions, and existing atomicity techniques. Section 3 highlights the design of SSP. Section 4 describes the architecture of SSP in detail. Section 5 evaluates our design and presents our key findings. Section 6 discusses related work. Section 7 concludes.

2 BACKGROUND AND MOTIVATION

In this section, we motivate our design by discussing the background on persistent memory and review the state of the art of existing failure-atomicity mechanisms.

2.1 Persistent Memory Systems

In the legacy model, system calls such as `read()` or `write()` have been used to operate on data stored in application buffers. System calls such as `fsync()` are used to write data back to the storage mediums. Our work focuses on a new NVRAM-style programming model [2, 4, 14, 16, 49] in which applications directly access the storage media (*e.g.* persistent memory) using processor load and store instructions while ensuring durability with instructions such as `cache-line-write-back`. The emerging memory technology and NVM-style programming model offers a number of unique opportunities to reduce the cost of persisting data. First, it enables us to leverage the byte-addressable representation to make data persistence fast. Second, we can leverage the virtual memory indirection to build an efficient address remapping scheme.

2.2 Requirement of Persistent Memory Transactions

The database community has defined four transaction properties: Atomicity, Consistency, Isolation, Durability (ACID). We use these properties to define the requirements for a system enabling failure-atomic transactions. Failure-atomic transactions may consist of multiple persistent writes. Partially completed transactions caused by abnormal termination need to be rolled back as they may lead to the violation of application semantics. Persistent writes within a transaction should be performed in an "all or nothing" fashion. This is referred to as **atomicity**. **Consistency** is application-specific and requires that updates performed by transactions (user-defined) always advance the system from one consistent state to another. In a multi-core environment, concurrent threads should never expose updates from incomplete transactions to each other. This is referred to as **isolation**. Finally, **durability** requires that all updates reach the non-volatile storage (*e.g.* NVRAM) before a transaction is acknowledged. This is a non-trivial requirement as persistent memory systems are likely to continue to support volatile caches (and possibly DRAM) for performance reasons. Flush operations can be utilized to force write back of dirty data from caches to persistent memory.

This work focuses on enforcing ACD properties. Programmers need to use locking or transactional memory [12, 43] to ensure the isolation of concurrent threads. While transactional memory might be combined with failure-atomic transactions, we leave this out for future work.

2.3 Drawbacks of Existing Crash-Atomicity Techniques

Storage systems traditionally ensure crash-consistency by using one of three techniques: write-ahead logging [32], shadow paging [13],

Name	Low Extra Writes	Low Persistence Overhead	Low Instruction Overhead
Software redo/undo logging	✗	✗	✗
ATOM, Proteus	✗	✗	✔
DHTM	✗	✔	✔
LSNVMM	✔	✔	✗
SCSP	✗	✗	✗
SSP	✔	✔	✔

Table 1: Summary of existing failure-atomicity mechanisms.

or log-structuring [40]. We now examine the use of these techniques in persistent memory transactions. Table 1 presents a summary of existing failure-atomicity mechanisms which we will discuss in more detail in the following.

Write-Ahead Logging. Whenever data is overwritten, either the original data or the new data must first be written to a logging area in NVRAM. Only after persisting the log to NVRAM, data can be updated in-place which guarantees that on a failure the original data can always be recovered. Different variants of logging implementations [4, 16, 18, 19, 44, 49] have been investigated by both academia and industry. All of these implementations share a common "write twice" problem, as NVM writes need to be performed twice: once to the log and once to the actual data. Prior approaches differ in terms of the techniques being used to minimize the software overhead as well as in how they reduce the impact of logging on overall performance. Logging in software [4, 16, 49] entails significant instruction overhead: software-based undo logging has to use excessive ordering and flushing instructions to ensure the ordering between the log update and the data update; software-based redo logging requires all memory reads to be intercepted and redirected to the redo log to obtain the latest values. Hardware support [18, 19, 44], therefore, has been introduced to provide high-performance logging. ATOM [19] and Proteus [44] (hardware undo logging) move the log update out of the critical path of the atomic update by tracking the dependency of the log update and the data updated in hardware. DHTM [18] (hardware redo logging) improves upon previous solutions by decoupling the data update from the transaction commit: the process of writing back the modified cache lines to persistent memory can overlap with the execution of the non-transactional code following the transaction. However, even under DHTM, committing the redundant writes to NVRAM remains on the critical path and as a results may delay subsequent transactions reducing overall performance.

Shadow Paging. When shadow paging performs a write, it creates a new copy of a memory page, updates the new copy, and then atomically updates the persistent virtual-to-physical address mapping to complete a failure atomic write sequence. BPFS [5] presents a redesign of traditional shadow paging for the NVRAM-aware filesystem, called Short-Circuit Shadow Paging (SCSP). SCSP includes two optimizations: i) it only copies the portions of the data in a page that will not be changed and ii) it applies 8-byte atomic updates as soon as possible to avoid propagating the CoW to the root of tree. While SCSP might be suitable for file system workloads where the file data updates tend to be large, persistent memory accesses are performed on the byte-granularity, rendering the method unsuitable for persistent memory systems.

Log-Structuring. Log-structured stores [40] do not update data in-place, but instead, append newly written data to the end of a log. A continuously updated mapping table is used to map logical addresses to physical storage. The unique challenge in using log-structuring for persistent memory systems is that, to limit the overhead of the mapping table, mappings need to refer to large, fixed-size blocks of data. Persistent memory systems, however, operate on fine grained byte-sized granularity, which introduces fragmentation and garbage collection overheads when utilizing large block sizes. Log-Structured Non-volatile Main Memory (LSNVMM) [14] proposes a sophisticated tree-based remapping mechanism which allows the out-of-place update to be performed at varied granularities. The mapping of LSNVMM is implemented as a partitioned tree (*e.g.* an array of skip lists); a node cache (*e.g.* hashtable) is employed to reduce tree traversal. Despite all its optimizations, the capacity overheads for storing the mapping tables as well as the instruction overhead are still significant in a system that offers data accesses at sub-microsecond latencies.

3 SSP DESIGN

In this section we introduce the design of SSP. First, we introduce the programming model and then we discuss the basic concept of SSP. We then introduce two key techniques: i) metadata journaling and i) page consolidation. We conclude this section with a discussion.

3.1 Programming Model and ISA Extension

We adopt a programming model proposed by Mnemosyne [49], in which programmers use the language construct `atomic{...}` to define a failure atomic section (*e.g.* updates inside are persist in a all or nothing fashion), and use the `persistent` keyword to annotate pointers to persistent data. Furthermore, we extend the ISA with a pair of new instructions—`ATOMIC_BEGIN` and `ATOMIC_END`—to define the begin and the end of a failure-atomic section and a new instruction called `ATOMIC_STORE` to hint a store must be conducted in an atomic fashion. `ATOMIC_BEGIN` and `ATOMIC_END` act as a full memory barriers. The `ATOMIC_STORE` instruction adds the store address to the transaction's write set so that it can be flushed to NVRAM during commit. The compiler can be modified to translate these software interfaces (*e.g.* the `atomic` block and the `persistent` pointers) to the ISA instructions.

Note that our interface resembles the interface of Intel TSX [15, 50] where `XBEGIN` and `XEND` are used to indicate the beginning and the end of a transaction. Intel TSX is designed to replace locking as a new way to ensure thread synchronization, and provides no guarantee on durability. In future work, we will investigate integrating SSP with hardware transactional memory through which we can provide an unified interface for supporting ACID transactions. In this work, we assume the isolation of threads are guaranteed using locks: by grabbing locks before the operation is performed, other threads are prevented from observing intermediate states.

3.2 Shadow Sub-Paging

Conventional shadow paging suffers from the problem that it operates on full pages which means a cache line write requires CoW of an entire page. To address this challenge, we propose Shadow Sub-Paging, a technique that can track updates on a much finer granularity: cache lines. Shadow Sub-Paging draws inspirations from the PTM-select technique [3], with major extensions to support failure-atomicity for persistent memory.

SSP Abstraction. Shadow Sub-Paging (SSP) is a persistent-memory-optimized version of shadow paging. When SSP is used to perform atomic updates, each active virtual page is associated with a second physical page. A page is active as long as it is in the TLB; for inactive pages the two physical pages are consolidated into one for space efficiency. We refer to the original physical page as "P0" and the extra physical page as "P1". Besides a second physical page, SSP requires three bitmaps for pages that are being actively updated. Specifically, the state of each cache line in the virtual page is represented by a single *current bit*, single *updated bit* and a single *committed bit*; each bit in these bitmaps refers to the cache line of the same offset. As bitmap access is performed on the critical path, they are cached in the TLB as we will explain in the metadata section. The *current bit* defines whether the most recent version of some data referred to by a virtual address is currently mapped to physical page P0 or P1. The *updated bit* is set to one whenever the cache line is written, and is reset as part of the commit process. The *updated bitmap* represents the write set of a transaction. The *Committed bit* defines whether P0 or P1 currently contains the committed (old) version of the cache line.

As a transaction is being processed, reads are directed to the page determined by the *current bit*. When the cache line is written to for the first time in a transaction SSP performs three tasks atomically. First, the corresponding *updated bit* is set to track the line as part of the transaction's write set. Second, the write is applied to the cache line that resides on the "other" page, for instance, to P1 if the committed cache line is part contained in P0. Third, the *current bit* is inverted such that the most recent (but still transient) version of the cache line now points to the new page. Note that in SSP it is possible that for a specific virtual page, some cache lines are currently stored on P0 and some on P1. Whenever a write targets a cache line that is already in the write set, it simply updates the current cache line. To commit a failure-atomic transaction, SSP persists all cache lines in the write set by flushing them to NVRAM. Note that cache lines might have already been persisted during the transaction in case they were evicted from the cache. This is not a problem in SSP, even in the case of a power failure, as writes never overwrite committed data in place. Furthermore, as part of the commit sequence, the *updated bitmap* is cleared to atomically commit the speculative updates; Lastly, the *committed bitmap* is persisted so that in case of a system failure, the *current bitmaps* can be recovered.

The approach explained above suffers from the following problem. Consider a cache line for a virtual address that is currently mapped to P0. If the cache line is transactionally written, the update needs to be applied to P1, requiring a copy-on-write of P0 into P1 which is costly. The other option would be to cache both the P0 and the P1 cache line. However, this would virtually reduce the cache size by 2×. We address this issue by the following technique. Instead of performing CoW, we directly apply the write to the cache line, however, we atomically change the tag so that the line now maps to the "other" page. As the "old" line has already been flushed to NVRAM as part of a previous commit, this approach is safe.

Metadata Storage. The per-page bitmaps need to be checked (and updated) in the critical path. As in prior work [42], we extend the TLB hardware to cache extra metadata required by SSP. Specifically, we store the second physical page number, the *updated bitmap*, and the *current bitmap* in the TLB hardware. As shown in Figure 1, threads (cores) are required to use their own set of *updated bitmaps* to track the write-set of the on-going transaction so that they can commit (or abort) their modifications in isolation. To ensure a single view of shared memory, all threads share a *current bitmap* for a given virtual page. We will discuss how to ensure the coherence of *current bitmap* among cores in section 4.1.2. Our system must guarantee that data from previously committed transactions can always be retrieved after a power cycle. The per-page *committed bitmap* is durably stored in the NVRAM and is updated as part of the commit process.

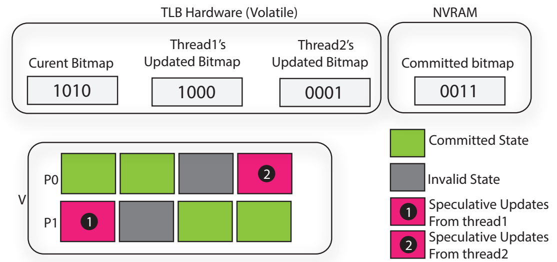


Figure 1: The metadata of SSP: each thread has to track their own write set with private *updated bitmaps*; all threads should agree on a *current bitmap* for a virtual page; the per-page *committed bitmap* is used to preserve the consistent state of a page and should be stored durably.

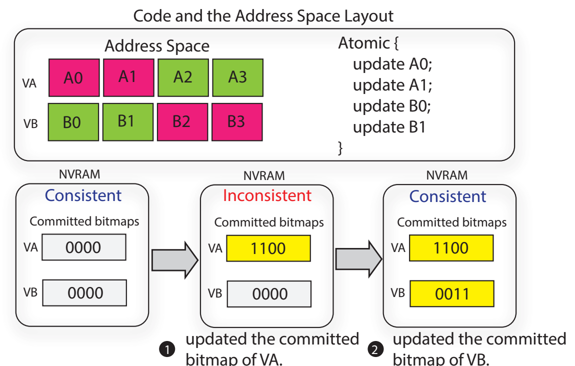


Figure 2: The consistency of SSP: the commit process might consist of updating multiple *committed bitmap* in the NVRAM; power failure in between will leave the system in an inconsistent state.

3.3 Metadata Journaling

To preserve the atomicity of data updates in the case of transactions spanning multiple pages, we must update all *committed bitmaps* **atomically** during transaction commit. An example describing this scenario is shown in Figure 2 where a code section specifies four cache lines need to be updated atomically. The commit process involves updating the *committed bitmap* of VA (from "0000" to "1100") and that of VB (from "0000" to "0011"). However, it might take two separate steps to update these *committed bitmaps* from the perspective of the memory controller. If the system crashes in between, only updates on VA (*e.g.* A0 and A1) will be visible after recovery, which violates atomicity. We use metadata journaling to ensure the atomicity of updates on the metadata of SSP. Our metadata journaling approach can be considered as a redo logging, however, only for the SSP metadata and not for the data itself as in conventional redo logging. It works as follows: every update to the per-page metadata, appends an entry (operation) to the log where an entry contains the page ID and the committed bitmap. Only after persisting the meta log for a transaction to NVRAM, SSP updates the per page committed bitmaps in the metadata area. More details on the implementation of metadata journaling are provided in section 4.1. As compared to data journaling (*e.g.* redo/undo logging), which requires to log every modified data block (*e.g.* typically 64 Byte), SSP journaling is lightweight as it only needs to record 128 bits of metadata for each modified page.

3.4 Page Consolidation

Associating each virtual page in the system with two physical pages represents a 2× capacity overhead. To address this problem, SSP consolidates physical pages into a single page whenever a virtual page is not actively used, and thus not contained in the TLB. As SSP requires pages to be resident in the TLB if they are written as part of a transaction, it is safe to consolidate a page even if some lines are still cached because a line without TLB mapping must either be committed or invalid.

At the time of consolidation, the valid data of a virtual page is likely to be distributed across the two associated physical pages. To minimize the data copying overhead, we identify the physical page (*e.g.* P0 or P1) which contains fewer valid cache lines and copy its valid data to the other physical page (*e.g.* P1 or P0). Note that we can easily compute the number of valid cache lines in P0 or P1 by counting the number of '0" or "1" in the corresponding *committed bitmap*. Finally, we update the virtual-to-physical mapping table so that the virtual page refers to the physical page with all the valid data.

Another issue that needs to be addressed is the accurate identification of inactive virtual pages. It is important, as premature consolidations of pages that are still being actively updated will result in unnecessary data copying overhead. We reuse the TLB hotness tracking: when a virtual page is not referenced by any TLB entry, we consider it as inactive. These inactive pages could be consolidated eagerly (*e.g.* immediately after being detected) or lazily (*e.g.* when the demands on the memory resources are high). Our current implementation consolidate inactive pages eagerly and we plan to investigate lazy consolidation in the future.

3.5 Discussion

Virtually-Indexed cache: SSP can work seamlessly with physical, or virtually-indexed physically-tagged caches. To support SSP on a virtually-indexed cache, we extend the tag with one *TX bit* to indicate whether a cache line has been modified by the current transaction. When a modified cache line is written back to memory, the *TX* bit allows Shadow Sub-Paging to distinguish transactional cache lines from regular cache lines. For a transactional cache line, we leverage SSP remapping to prevent overwriting the committed data. A read miss will also require to access the extended TLB. In this case, SSP locates the current mapping (P0 or P1) of a cache line.
Superpages: Superpages [7, 22, 34, 46] are commonly used to increase the coverage of the TLB. Supporting superpages in SSP is challenging due to the large per-page metadata overhead. For instance, a 2 MiB page has 32,768 cache lines and thus, the required bitmap size is 262,144 bytes. It is unpractical to scale TLB entries to support such large bitmaps. SSP currently only supports 4 KiB base pages. However, techniques such as transparent superpages and page clustering as supported by Linux can be extended to support SSP. In particular, coexistence of small and superpages is possible by automatically demoting superpages when they are updated and promoting pages to superpages when they become "inactive". With this approach, superpages can be used for read-only data. As for the design of TLB hardware, support for superpages and that for SSP can coexist because most processor vendors use split TLBs [7]. SSP only requires TLB extensions for the 4 KiB base pages.
Limitation and Fall-back path. The SSP design has limitations in terms of the size of a transaction it can support. If a transaction updates more pages than the TLB can hold, SSP needs to abort the transaction and revert to a fall-back path. The fall-back path transfers control to a programmer-defined handler which can implement any kind of unbounded software redo or undo logging to ensure atomicity. SSP is designed to handle small and medium sized transactions efficiently, similar to existing commercial HTMs [15, 50].

4 SSP ARCHITECTURE

Figure 3 depicts the architectural details of SSP. The per-page metadata of SSP, which contains persistent fields such as the *committed bitmap* and volatile fields such as the *current bitmap*, is managed by the memory controller in the form of the SSP cache. We extend the TLB hardware to cache the *current bitmap*, the *updated bitmap* and the second physical page number. The core is extended to handle atomic updates of cache lines so that on each write the cache tag, the *updated bit* and the *current bit* is updated atomically. To support transaction commit, the core needs to i) write back the cache lines that have been modified and ii) issue a metadata update instruction for each modified page to the memory controller. The memory controller performs metadata journaling to ensure the atomicity of metadata updates. Furthermore, the memory controller tracks the status of each page using the SSP cache and it also conducts page consolidation.

4.1 Architectural Extensions

4.1.1 Extensions on CPU hardware. We propose a set of architectural extensions to facilitate SSP. In particular, we adopt a wider TLB entry in which we can cache the second physical page number,

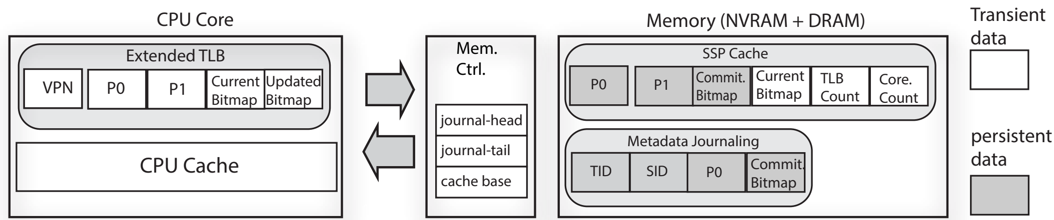


Figure 3: The architecture of SSP.

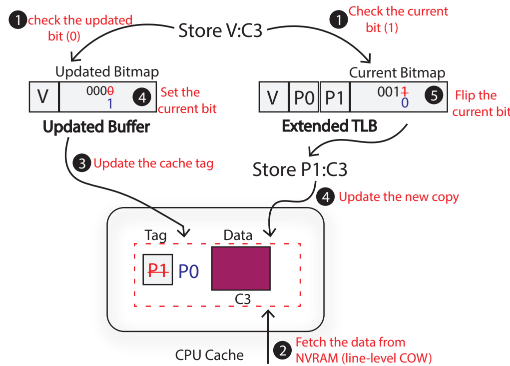


Figure 4: The atomic update process of SSP: update the fourth cache line of page V; step ❶ - ❷ generate a copy of data in cache; step ❸ - step ❻ complete the CoW.

the *current bitmap* and the *updated bitmap* of a page that is being modified. In the event of TLB miss, the core will conduct a page table walk as usual to obtain the original physical page number (*e.g.* P0) for the missing page. Afterwards, it interacts with the memory controller to fetch the SSP-specific metadata (using P0 as index), that is the second physical page number (*e.g.* P1) and the *current bitmap*; the updated bitmap is initialized with all zeros.

Memory Read and Write. During address translation, the virtual address is translated into either P0 or P1, depending on the corresponding *current bit* for the accessed cache line. The remainder of the memory access path remains unmodified.

Atomic Update. Figure 4 shows how SSP handles atomic updates. We assume a write-back cache with a write allocate policy. An atomic update process is described as follows: i) Shadow Sub-Paging checks the *current bit* to determine which page to write; ii) a copy of the data is loaded into the cache if it is not present; iii) the cache line is remapped by changing the tag, so the "other" cache line can be updated; iv) the new cache is updated with the written data; v) the *current bit* is flipped. Note that iii) and v) are only necessary if the line was written for the first time during the transaction. Since this modified cache line now is associated with another page, it is safe to evict it from the cache anytime without worrying about overwriting the committed state in NVRAM.

To conduct the remapping, the *current bitmap* for a page needs to be changed. To keep the current state of shared pages coherent across cores (and the memory controller), the most straightforward solution is to perform a TLB shootdown. However, TLB shootdowns incur significant overheads [1, 48]. We instead adopt the approach proposed by page-overlays [42] which exploits the cache coherence network to guarantee coherency of the TLB entries, including the *current bitmap*. The cache coherence network is extended with a new message called *flip-current-bit*. When a cache line is updated for the first time in a transaction (*current bit* is zero), a *flip-current-bit* message is broadcast via the cache coherence network to notify other cores as well as memory controllers to flip the *current bit* for the corresponding cache line. Note that we may piggy back the *flip-current-bit* on the invalidation message. The approach can be trivially extended to support directory based cache coherency protocols. We believe the broadcasting will only impose minimal overhead on the system overall. As shown in previous work [33], in typical PM workloads, only a small portion (< 4%) of accesses are to PM; the majority to DRAM. Moreover, our design only requires a broadcast operation for a fraction of stores to PM (e.g. modifying a cacheline for the first time in a transaction).

Transaction Commit. Durable transactions must guarantee data persistence after commit requests are acknowledged. The commit process of SSP involves two steps i) data persistence and ii) metadata update. Atomic updates themselves do not ensure data persistence—some updates might still be in the cache at the transaction commit. Here we use a write-back instruction such as *clwb* to write back the cache lines modified by the committing transaction. The write-set of the committing transaction is tracked by the *updated bitmaps* stored in the extended TLB. The commit process also needs to update the *committed bitmaps* stored in NVRAM. We extend the write interface of the memory controller with a special metadata update instruction. For each updated page (identified by the update buffer), we pass information such as the page ID (*e.g.* P0) and the *updated bitmap* to the memory controller using the metadata update instruction. The memory controller will perform journaling to ensure the atomicity of the metadata updates. The metadata update instructions are passed to the memory controller without caching. Note that we must ensure the ordering between the data persistence and the metadata update. If the system crashes before the atomic metadata update is complete, all speculative updates will be discarded, recovering into an consistent state.

4.1.2 Memory Controller Extensions. In the SSP architecture, the memory controller provides centralized storage for metadata and it performs page consolidation and metadata journaling. Furthermore, it is responsible for managing a set of pages from which the second physical pages can be allocated. Note that the number of pages is

bounded by the number of TLB entries and that page consolidation ensures that pages will be freed after they become inactive.

Metadata Storage. SSP associates each virtual page with additional metadata. Since we only need these additional fields when pages are active, we do not require extension of the page table entries. Instead, the memory controller maintains a SSP cache separately to store the SSP-related metadata. An SSP cache entry contains the following details regarding a page that is being actively updated: the original/second physical page number (PPN0/PPN1); the consistent state (*committed bitmap*); the current state (*current bitmap*); the number of TLBs that have cached the translation for this page (TLB reference count); the number of cores that are updating this page (*core reference count*). Among this information, the physical page numbers and the *committed bitmap* must be stored durably. Fields such as the *current bitmap*, the core/TLB reference count are transient and are not necessary for the recovery.

Whenever a SSP cache entry is accessed (*e.g.* after a TLB miss), the SSP cache is consulted with the original physical page number (P0). In case of a miss, the memory controller inserts a new entry into the SSP cache. The replacement algorithm of the SSP cache is straightforward. The memory controller may evict any entry that contains a page that is i) already consolidated (*e.g. committed bitmap* is zero) and ii) not referenced by any TLB (*e.g.* TLB reference count is zero). The SSP cache can be sized according to the TLB and the number of cores. For instance, in a system with N cores and T-entry TLBs, the size of the SSP cache is set to $N \times T + O$. Here O is the overprovisioning factor used to accommodate pages that are being consolidated. The rationale behind this is that i) the maximum number of concurrent transactions is N, ii) each transaction can touch no more than T pages, and iii) O entries are overprovisioned so that we do not have to wait for pages to be consolidated in order to make room for new TLB-fill requests. If under rare conditions, we find that the cache entries we reserve are not enough, we can resize the SSP cache and request more pages from the OS.

Free Space Management. At system initialization, the OS will reserve a small mount of continuous NVRAM physical pages and pass the base address to the memory controller by setting one of its registers. The memory controller will associate each entry of the SSP cache with an extra physical page up front. The extra physical page is utilized by the virtual page assigned to an entry and can be reused when the entry is assigned to a new virtual page as all data stored in the extra page is persisted during consolidation. To overcome uneven wear out, the memory controller may exchange the per-slot extra physical pages with fresh pages from time to time.

Page Consolidation. SSP decides whether a page is eligible for consolidation according to the following information: is there any TLB that has cached the SSP cache entry for this page? Specifically, the TLB reference count is used to decide when to consolidate a page. The TLB reference count is increased by one if a core fetches the SSP cache entry from the memory controller and is decreased by one if a core evicts the SSP cache entry from its TLB. When the memory controller detects that the reference counter for a page drops to zero, an entry, which includes the two physical page numbers and the *committed bitmap*, is inserted into a consolidation queue. An OS thread conducts page consolidation in the background, allowing the new TLB entry to be inserted with minimal delay. When a page has been consolidated, the consolidation thread inserts an entry into a finish queue to notify the controller. We reserve several bits in the SSP cache entry to track the status of a page (e.g. whether it is being consolidated). In the rare case a page is requested by the TLB during consolidation, the response is delayed until after the consolidation for that page has been completed.

Metadata Journaling. A multi-page transaction requires multiple updates to the metadata area that stores the pages' *committed bitmaps*. As shown in Figure 3, each metadata journaling record, which represents the intention to update the SSP cache, has four fields—the Transaction ID (TID), the ID of the cache slot that that is being modified (SID), the new value of original physical page number and the new value of the *committed bitmap*. The TID is assigned by the memory controller to uniquely identify the metadata updates from the same transaction. The SID is used to compute the physical address of the slot given the base address of the SSP cache. Upon receiving a metadata update instruction, the memory controller generates a record and appends it to the metadata journal. Note that journaling records are written back to NVRAM, at cache level granularity, only when the log buffer is full or an explicit request is made to flush the buffer.

Checkpointing. To limit the growth of the journaling space and also to bound the recovery time, the memory controller needs to perform checkpointing, which updates the state of the persistent SSP cache to the most recent consistent snapshot and then needs to clear the journaling space. A background OS thread is used for checkpointing and takes three steps: i) it records the current head pointer—where appends happen—of the journal, ii) it applies the log records to the persistent SSP cache and iii) it advances the tail pointer of the journal. Note that the checkpointing thread will capture the final state of a modified cache entry and only write it back to the persistent cache.

4.2 Architecture Details

We here discuss several optional details that improve the efficiency of the implementation of SSP.

Write-set Buffer. Storing the *updated bitmap* in the TLB entry along with the physical page numbers and the *current bitmap*, albeit straightforward, entails a problem—the burst of non-transactional accesses may cause an in-transaction TLB entry (*e.g. updated bitmap is non-zero*) to be evicted, making it impossible to track the write-set of the transaction. To address this issue, a separate write-set buffer can be added to store the *updated bitmaps*. The write-set buffer is cleared once the ongoing transaction is committed. By decoupling the *updated bitmap* from the TLB, a page might be evicted from the TLB while it is written as part of a ongoing transaction. We deal with this corner case with a per-page core reference count. The per-page reference count is increased upon receiving a *flip-current-bit* from a specific core, and is cleared upon receiving a metadata update instruction. A page with non-zero core reference count will not be considered for consolidation or cache eviction.

SSP Cache Organization. Transient runtime information such as the reference count is updated frequently. Placing the SSP cache in NVRAM will cause unnecessary wear out. The SSP cache is organized as a transient SSP cache (stored in DRAM) and a persistent SSP cache (stored in NVRAM): the transient cache is employed to serve the requests from the cores; the persistent cache serves as a

backup and is used only during recovery. We only store persistent metadata such as the physical page numbers and the *committed bitmap* in the persistent cache.

To leverage faster memory in the hierarchy, we use a small portion of the L3 to as a "cache" for the SSP cache [6]. Only 1% of a 12-megabyte L3 cache could be used to cache about 4K SSP cache entries. We will study the sensitivity of the access latency of the SSP cache in the evaluation.

4.3 Hardware Cost and Complexity Trade-off

There are two main hardware overheads in our design: the extended TLB entries and the write-set buffer. For a typical 4 KiB page and 64 byte cache line, there are 64 cache lines per page, so each bitmap has 64 bits, adding 64 bits and a second physical page number (*e.g.* 40 bits) to each TLB entry. Across the 64-entry L1 TLB, the overall cost to expand the TLB is 832 bytes. If we take the L2 TLB into consideration, a 1024-entry L2 TLB will add another 13 kilobytes. Each write-set buffer entry includes a 36-bit tag and a 64-bit bitmap. The size of a 64-entry write-set buffer is therefore 800 bytes. Thus, the overall hardware cost is 14.6 kilobytes.

We now identify opportunities to address the hardware overhead. In our original design, we conservatively assume the ideal granularity for ensuring persistence is 64 bytes (e.g. cache line granularity). However, as disclosed by a recent work [17], the preferable granularity for persisting data to the Intel's Optane DC Persistent Memory is 256 bytes. Utilizing 4× larger sub-pages, the size of the bitmap could be reduced to 16 bits, significantly reducing state overhead of the TLB entries. Furthermore, recent Intel, IBM and ARM processors provide HTM support. HTM uses one transactional bit per cache line to track the speculative updates. By reusing the transactional bit, we might be able to eliminate the need for using the *updated bitmaps*.

Our current design trades-off complexity for higher performance. To reduce hardware complexity, we may drop the modifications on TLB hardware by implementing SSP mappings in userspace. However, this imposes significant instruction overheads as now every load/store must be intercepted similarly to software transactional memory systems. Second, we can avoid the changes on the TLB coherence network by using TLB shootdowns instead. However, the TLB shootdown procedure involves trapping into the OS, issuing inter-process interrupts, imposing a significant performance overhead. Note that our current implementation only serves as a baseline for exploring the viability of SSP. Other alternatives might be considered in practice. Prior work [17] also discloses that Optane Persistent Memory embeds an address indirection table to achieve wear leveling and bad-block management. Such an indirection layer offers opportunities to incorporate the functionalities of SSP entirely inside the PM controller, significantly reducing the complexity without sacrificing much performance.

4.4 Recovery

Upon restart from an unclean shutdown, SSP performs the following two steps for recovery. First, it rebuilds the transient SSP cache with the persistent metadata stored in the persistent cache. Specifically, fields such as the two physical page numbers and the *committed bitmap* are reloaded directly from the persistent cache. Then the *current bitmap* is initialized with the value of the *committed bitmap* all other transient fields (*e.g.* reference counters) are initialized as zeros. Furthermore, the state of the transient cache needs to be updated to the most recent consistent snapshot, replaying the records in the metadata journal whereas the entries of aborted transactions are skipped.

Component	Parameters
Processor	4 OoO Cores, 3.7 GHz, 5-wide issue, 4-wide retire, 128 ROB entries, Load/Store Queue: 48/32, 64 DTLB entries
L1I and L1D	32 KiB, 64-byte lines, 8-way, 4 cycles
L2	256 KiB, 64-byte lines, 8-way, 6 cycles
L3	12 MiB, 64-byte lines, 16-way, 27 cycles
DRAM	8 GiB, 1 channel, 64 banks per rank, 1 KiB row-buffer, read/write 50 ns
NVRAM	8 GiB, 1 channel, 32 banks per rank, 2 KiB row-buffer, read/write 50/200 ns

Table 2: System Parameters

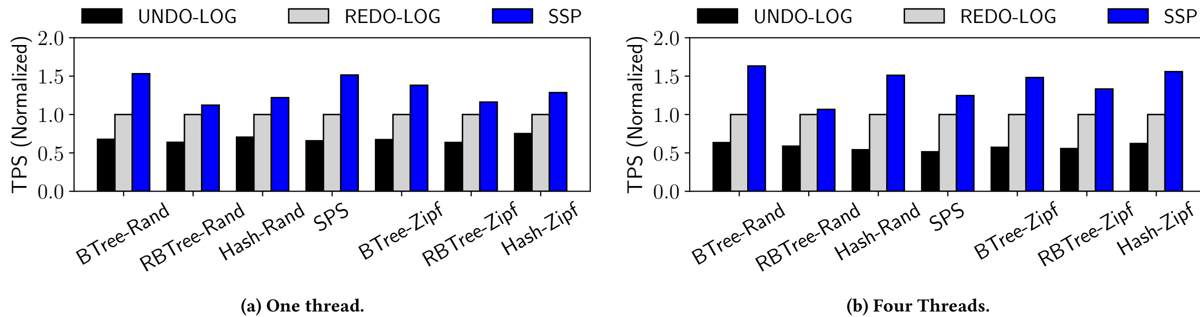
Name	Write Set	Description
RBTree-Rand	12/3/13	Insert/delete nodes in a red-black tree; Random workloads
BTree-Rand	10/6/21	Insert/delete nodes in a B+-Tree; Random workloads
Hash-Rand	3/3/4	Insert/delete nodes in a hashtable; Random workloads
SPS	2/2/2	Swap elements in an array
RBTree-Zipf	5/2/6	Insert/delete nodes in a red-black tree; Zipfian workloads.
BTree-Zipf	6/4/15	Insert/delete nodes in a B+-Tree; Zipfian workloads.
Hash-Zipf	3/3/4	Insert/delete nodes in a hashtable; Zipfian workloads
Memcached	3/2/35	Memslap as workload generator; Four clients; 90% SET
Vacation	4/3/9	Four clients; 16 million tuples

Table 3: A list of evaluated microbenchmarks showing the write set size (average number of cache lines modified / average number of pages modified / maximum number of pages modified). The write set consists of atomic updates within a transaction.

5 EVALUATION

5.1 Experimental Setup

We implemented SSP on `MarssX86` [37], which is a cycle-accurate full system simulator for the x86-64 architecture. We integrated `DRAMSim2` [41] into `MarssX86` for a more detailed memory simulation. The `DRAMSim2` is extended to model a hybrid memory system with both DRAM and NVRAM connected to the same memory-bus. Table 2 shows the main parameters of the system. Our simulated machine supports out-of-order execution, and includes a 64-entry L1 DTLB, 3 levels of cache, and an NVDIMM.



(a) One thread. **(b) Four Threads.**

Figure 5: Performance of micro-benchmarks (higher is better).

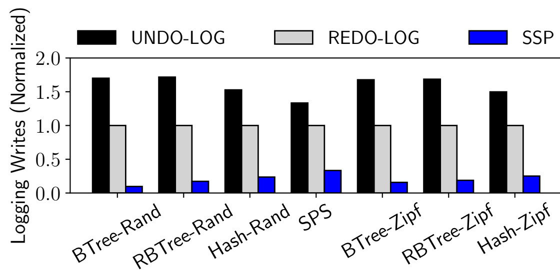


Figure 6: Comparison of logging writes (lower is better).

Simulation Methodology. We simulate the impact of logging, page consolidation as well as data persistence using the `MarssX86` and `DRAMSIM2`. To model the impact of SSP on the TLB and on cache coherency, we measure the number of TLB misses and the number of *flip-current-bit* messages. Note that we only count the TLB misses caused by accessing the persistent heap. The latency of accessing SSP cache is modeled for a given workload according to the L3 SSP cache miss ratio, L3 latency (*e.g.* 27 cycles) and DRAM latency (*e.g.* 185 cycles). The extra cycles are then added to the total cycles. To better understand the impact of the latency of SSP cache access, we conduct a sensitivity study in Section 5.3. In our experiment, we reserve 0.3% of the L3 cache to be used to store the SSP cache (*e.g.* about 1K SSP cache entries).

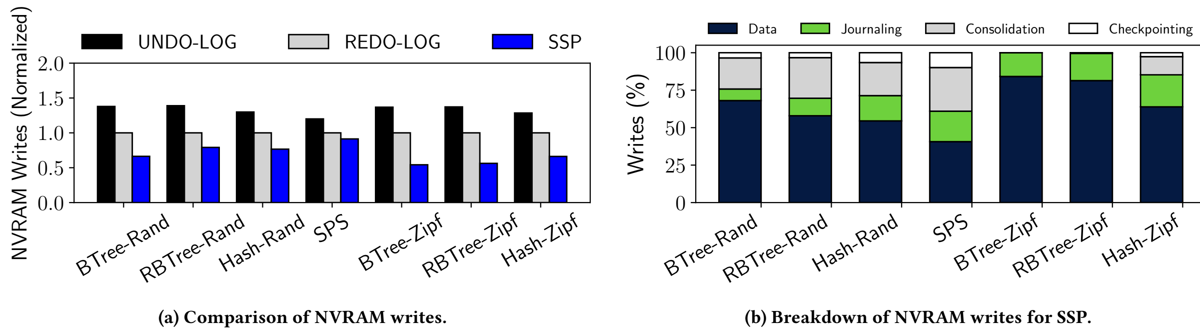
Benchmarks. We evaluate both microbenchmarks and real workloads in our experiments. The microbenchmarks cover commonly used data structures such as the B+-Tree (*BTree*), as described in Table 3. The elements, keys or values used in these workloads are all 8-byte integers. Each data structure update (*e.g.* insert, delete, or swap) is wrapped inside a durable transaction. Benchmarks *BTree*, *RBTree* and *Hash* first search for a key and then either delete (key found) or insert (key absent) a key/value pair. We vary the access patterns for these workloads by changing the key distribution. We use suffix "-Rand" and "-Zipf" to denote random workloads and zipfian workloads. For zipfian workloads, 80% of the updates are applied to 15% of the keys. The key/value pairs are generated prior to each run. We evaluate two real applications: Memcached [10] is a well-known in-memory Key/Value cache and Vacation [31] which emulates an OLTP system. Prior work [33] has published the persistent-memory-aware version of these applications; we merely replace their durable interfaces with ours. The characterization of the benchmarks is shown in Figure 3. As none of the evaluated applications writes to more than 64 pages during a transaction, a 64-entry write-set buffer is sufficient to accommodate all of the workloads. As a result, none of our evaluated applications requires the unbounded fall-back path.

Evaluated Designs. We compare SSP with two other designs for which we use tuned, optimal parameters (*e.g.* size of the log buffer). We do not compare with conventional shadow paging. As shown in Table 3, transactions only touch 2-6 cache lines on average. Conventional shadow paging degrades performance by writing up to 64× more cache lines.

- UNDO-LOG represents a naive hardware undo logging mechanism. Each atomic store will generate a log entry. The store then will be blocked until the log entry reaches persistent memory. Under undo logging, if a value is repeatedly updated multiple times, we only need to generate a log entry for the first update. We employ a log buffer to avoid writing redundant log entries.
- REDO-LOG [18] is a state-of-the-art hardware redo logging. It allows overlapping data persistence with the non-transactional code following the transaction commit. Besides, it also employs a log buffer to predict the final state of a cache line and thus avoids redundant log entries.

5.2 Mirobenchmark Results

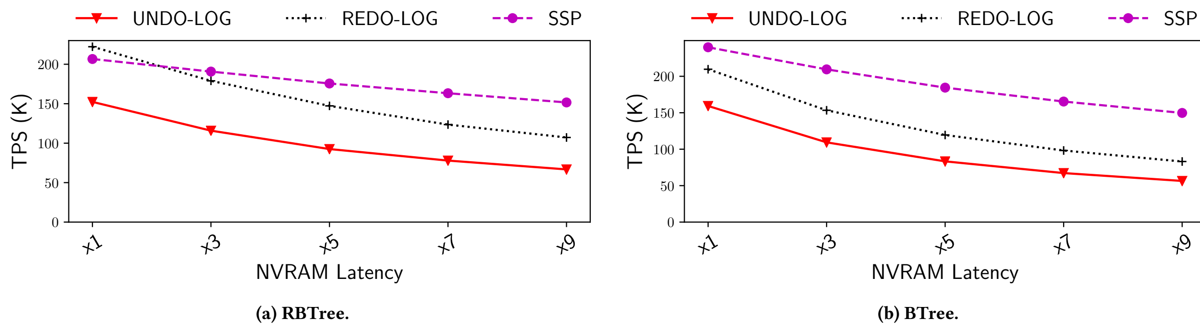
Transactional throughput. We show performance results of the four designs running the microbenchmarks. From Figure 5a, we observe that SSP outperforms UNDO-LOG and REDO-LOG by 1.9× and 1.3× on average under single threaded workloads. The improvement mainly comes from the ability of SSP to reduce the logging overhead. As shown in Figure 6, SSP can decrease the write traffic caused by logging by 7.6× and respectively 4.7× compared to UNDO-LOG and UNDO-LOG. In particular, under the *BTree-Rand* workload, SSP nearly eliminates the logging writes, which results in a 1.5× improvement in transactional throughput. As we can see in Table 3, the *BTree* benchmark exhibits great spatial locality (*e.g.* several cache lines modified in a single page), which minimizes the writes introduced by metadata journaling. Figure 5a shows the performance under four threads. We can see SSP scales well. SSP can



(a) Comparison of NVRAM writes.

(b) Breakdown of NVRAM writes for SSP.

Figure 7: NVRAM writes.



(a) RBTree.

(b) BTree.

Figure 8: Sensitivity to the Latency of NVRAM: the x-axis shows the NVRAM latency in multiple of DRAM latency.

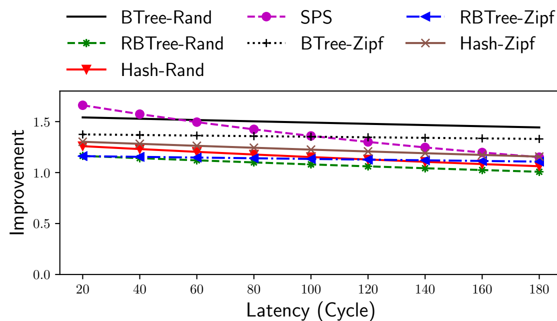


Figure 9: Sensitivity to the latency of SSP Cache: y-axis shows the speedup over REDO-LOG.

improve the performance by 2.4× and 1.4× over the UNDO-LOG and REDO-LOG on average, respectively.

NVRAM Writes. Figure 7a compares SSP to the baseline designs in terms of the number of NVRAM writes. We make two observations. First, SSP can save 45% and 28% write traffic as compared to UNDO-LOG and REDO-LOG on average, as the Undo/Redo logging designs essentially require data to be written twice. Although page consolidation also causes extra writes in SSP, it is not a per-transaction operation: modified data does not require an immediate copy operation during the transaction commit but instead additional writes are only required when an active page turns inactive. As the transaction frequency is much higher than the frequency of pages becoming inactive, SSP can effectively "batch" the additional writes required for failure-atomicity. Figure 7b shows the breakdown of NVRAM writes in our SSP design. As we can see, the number of writes caused by page consolidation is less than the data writes under most of the workloads except for *SPS*. Second, the locality of the workloads affects the number of NVRAM writes. Under benchmarks with zipfian access pattern (*e.g. BTree-Zipf, RBTree-Zipf* and *Hash-Zipf*), SSP on average can reduce the write traffic by 56% and 42% over UNDO-LOG and REDO-LOG. In contrast, for random workloads using a unified distribution SSP can only save 43% and 23% write traffic over UNDO-LOG and REDO-LOG. As shown in Figure 7b, under workloads with locality, extra writes caused by page consolidation are negligible. It demonstrates the SSP design can efficiently prevent the premature consolidation of hot pages and thus minimize page consolidation overhead for zipfian workloads.

5.3 Sensitivity Study

Latency of NVRAM. Figure 8 shows the transaction throughput with varying memory latency. For brevity, we only show the results of two representative workloads here. Overall, it can be seen that the performance of SSP and the baseline designs degrade when the NVRAM latency increases. However, the gap between SSP

	UNDO-LOG	REDO-LOG
Memcached	75%	35%
Vacation	27%	13%

Table 4: The performance improvement over other designs for Benchmarks *Memcached* and *Vacation*.

	UNDO-LOG	REDO-LOG
Memcached	49%	46%
Vacation	38%	17%

Table 5: The saving of write traffic over other designs for Benchmarks *Memcached* and *Vacation*.

and other designs is increasing as well. In particular, the speedup over REDO-LOG increases from 1.1× to 1.8× under the benchmark *BTree* (Figure 8b). SSP minimizes the logging writes and thus is less sensitive to the change of the NVRAM latency. We observe that when the NVRAM is as fast as DRAM, REDO-LOG outperforms SSP by 8% under the benchmark *RBTree* (Figure 8a). The reason behind this is that when the persistency overhead is low (*e.g.* DRAM latency), REDO-LOG can hide the most of the delay caused by persisting data.

Latency of the SSP Cache. Figure 9 shows the impact of the latency of the SSP cache on the performance of our SSP design. For all workloads besides *SPS*, the cache latency has limited impact on SSP performance, showing only a moderate linear performance decrease with increased latency. However, the latency of SSP cache is still critical for benchmarks such as *SPS* and *Hash-Rand*. This is because their poor locality leads to frequent TLB misses, which in turn increase the frequency of accessing the SSP cache. We observe that the zipfian workloads are less sensitive to the latency of the SSP cache than these random ones. This can also be explained by the difference in locality exposed by these workloads.

5.4 Performance of Real Workloads

Table 4 shows the performance improvement of SSP over other designs for the *Memcached* and *Vacation* benchmarks. For the *Memcached* benchmark, SSP provides a 74% throughput improvement over UNDO-LOG and a 35% higher throughput compared to REDO-LOG. For the *Vacation* Benchmark, SSP provides a 27% improvement over UNDO-LOG and 13% higher throughput over REDO-LOG. The improvement comes from the reduction in logging overhead. Specifically, SSP saves 86% and 82% logging writes over UNDO-LOG and REDO-LOG under the real workloads on average. In the *Vacation* benchmark, SSP generates less improvement over REDO-LOG. This is because the volatile execution contributes to most of the overhead of the *Vacation* benchmark.

Table 5 shows the reduction of NVRAM writes. As we can see, SSP continues to save write traffic to NVRAM: 49% and 46% reduction over UNDO-LOG and REDO-LOG under the *Memcached* Workload and 38% and 17% reduction over UNDO-LOG and REDO-LOG under the *Memcached* Workload. The extra write traffic caused by page consolidation is only 15% and 31% of the total write traffic for the *Memcached* and *Vacation* workloads.

6 RELATED WORK

NVRAM-aware data structures [25, 47, 53] focus on reducing persistence costs for particular data structures. In contrast, SSP support atomic and durable updates of *any* data structure. Existing NVM-support libraries [4, 16, 49] and NVRAM-aware file systems [5, 9, 51, 52] use either undo/undo logging or shadow paging to build durable transactions. SSP can increase the efficiency of these existing approaches.

Prior work [14, 30] has proposed optimizations for software-based failure-atomicity mechanisms. Both Kamino-TX [30] and LSNVMM [14] attempt to reduce the persistency overheads by eliminating the extra writes in the critical path. Kamino-TX maintains a separate backup when modifying data, and LSNVMM uses a log-structured approach. However, both of the techniques have inefficiencies. LSNVMM introduces significant instruction overhead by maintaining an additional level of indirection in userspace, while Tamino-TX may delay dependent transactions (*e.g.* read/write set overlap with prior transaction). DudeTM [28] enjoys the benefits of a redo-log (fewer CPU flushes/barriers) while avoiding the drawbacks of an address remapping approach. However, DudeTM still must log the actual data modifications and apply updates to persistent storage afterwards. In comparison, SSP requires no costly software mapping, imposes no delay on the dependent transactions and minimizes extra writes on the critical path.

Several recent studies [8, 18, 19, 36, 44] propose hardware support to reduce the overheads of logging. However, none of these designs address the "write twice" problem introduced by logging and, thus, still suffer from the performance degradation caused by extra write traffic in the critical path. Recent studies [11, 21, 29, 38] also propose hardware/language-support for relaxing the ordering of NVM writes. LOC [29] proposes architectural modifications to relax the intra-transaction and inter-transaction ordering. Kolli et al. [21] propose optimizations to implement durable transactions based on relaxing memory persistency [38]. These proposals are orthogonal to our study and could be applied to further improve our design. Several designs [44, 54] propose leveraging non-volatile caches and write protected queues in memory controller to reduce logging-introduced persistency overhead. Our design can trivially be extended to take advantage of these advanced hardware features. Our previous work [35] introduces the basic concepts of Shadow Sub-Paging with preliminary design, and early-stage performance evaluation. In this paper, we go beyond it in terms of techniques and evaluations.

Page overlays [42] aim to provide fine-grained memory management with a per-page bitmap. It proposes a wider TLB entry to track the updates at the cache line level. However, page overlay semantics don't allow us to build durable transactions efficiently, since page consolidations are required for every transaction commit on the critical path. SSP is designed to provide transactional NVRAM updates efficiently. In particular, SSP only needs to perform page consolidation upon TLB evictions and performs consolidations out of the critical path. SSP preserves the consistency of metadata that is required for recovery leveraging lightweight metadata journaling. SI-TM [27] and EXCITE-VM [26] leverages the indirection of virtual memory to build more efficient snapshot isolation transactions. However, it does not address the durability issues for persistent

memory transactions. PTM [3] leverages virtual memory to support unbounded transactional memory and presents a similar cache line level mapping semantic to reduce page level copying. However, our work goes beyond this work by addressing two additional challenges: First, we enable failure-atomicity including metadata updates and second we introduce page consolidation to address the 2× space overhead of PTM.

7 CONCLUSION

In this paper we proposed SSP, a novel shadow paging scheme that leverages fine grain cache line level remapping, to enable efficient, failure-atomic transactions. In particular, SSP eliminates most of the redundant writes introduced by prior log-based techniques. Our key idea is that we can delay the application of redundant writes via address remapping, enabling write batching to reduce the overall number of writes to NVRAM. By introducing cache line remapping, our technique successfully eliminates the copy-on-write overhead that made prior shadow mapping schemes unfeasible, while only requiring moderate changes to the TLB hardware. In addition to improving endurance, SSP removes redundant writes from the critical path improving transactional performance. In particular, our experimental results show that SSP can reduce overall NVRAM writes by up to 1.8×, and improve performance by up to 1.6×, as compared to a state-of-the-art hardware logging.

ACKNOWLEDGEMENTS

We would like to thank anonymous reviewers for their insightful comments. This research was supported by the NSF grants IIP-1266400, IIP-1841565, #1829524, #1829525, #1817077, #1823559, the industrial sponsors of the Center for Research in Storage Systems, and SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory.

REFERENCES

[1] Nadav Amit. 2017. Optimizing the TLB shootdown algorithm with page access tracking. 27–39.

[2] Daniel Bittman, Peter Alvaro, Darrell D. E. Long, and Ethan L. Miller. 2019. A Tale of Two Abstractions: The Case for Object Space. In *Proceedings of HotStorage '19*.

[3] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. 2006. Unbounded page-based transactional memory. *ACM Sigplan Notices* 41, 11 (2006), 347–358.

[4] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. 105–118. http://www.ssrc.ucsc.edu/PaperArchive/coburn-asplos11.pdf

[5] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-Addressable, Persistent Memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*. Big Sky, MT, 133–146. http://www.ssrc.ucsc.edu/PaperArchive/condit-sosp09.pdf

[6] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. 2010. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE micro* 30, 2 (2010), 16–29.

[7] Guilherme Cox and Abhishek Bhattacharjee. 2017. Efficient Address Translation for Architectures with Multiple Page Sizes. In *Proceedings of the 2017 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 435–448.

[8] Kshitij Doshi, Ellis Giles, and Peter Varman. 2016. Atomic persistence for SCM with a non-intrusive backend controller. In *Proceedings of the 22th Int'l Symposium on High-Performance Computer Architecture (HPCA-22)*. IEEE, 77–89.

[9] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys '14)*. http://www.ssrc.ucsc.edu/PaperArchive/dulloor-eurosys14.pdf

[10] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.

[11] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M Chen, and Thomas F Wenisch. 2018. Persistency for synchronization-free regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 46–61.

[12] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA '93)*. ACM, New York, NY, USA, 289–300. https://doi.org/10.1145/165123.165164

[13] Dave Hitz, James Lau, and Michael Malcom. 1994. File System Design for an NFS File Server Appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*. San Francisco, CA, 235–246. http://www.ssrc.ucsc.edu/PaperArchive/hitz-usenix94.pdf

[14] Qingda Hu, Jinglei Ren, Anirudh Badam, and Thomas Moscibrod. 2017. Log-Structured Non-Volatile Main Memory. In *Proceedings of the 2017 USENIX Annual Technical Conference*. Santa Clara, CA, 703–717. http://www.ssrc.ucsc.edu/PaperArchive/hu-atc17.pdf

[15] Intel Corporation. 2012. Architecture instruction set extensions programming reference.

[16] Intel Corporation. 2015. Persistent Memory Programming. http://http://pmem.io/.

[17] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1903.05714* (2019).

[18] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2018. DHTM: Durable Hardware Transactional Memory. In *Proceedings of the 45th Int'l Symposium on Computer Architecture*.

[19] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. 2017. ATOM: Atomic durability in non-volatile memory through hardware logging. In *Proceedings of the 23th Int'l Symposium on High-Performance Computer Architecture (HPCA-23)*. IEEE, 361–372.

[20] T. Kawahara, K. Ito, R. Takemura, and H. Ohno. 2012. Spin-transfer torque RAM technology: Review and prospect. *Microelectronics Reliability* 52 (2012), 613–627. http://www.ssrc.ucsc.edu/PaperArchive/kawahara-mr12.pdf

[21] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the 2016 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. http://www.ssrc.ucsc.edu/PaperArchive/kolli-asplos16.pdf

[22] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. 2016. Coordinated and efficient huge page management with Ingens. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*. 705–721.

[23] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory As a Scalable DRAM Alternative. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA '09) (ISCA '09)*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1555754.1555758

[24] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*. 273–286. http://www.ssrc.ucsc.edu/PaperArchive/lee-fast15.pdf

[25] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems.. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*. 257–270.

[26] Heiner Litz, Benjamin Braun, and David Cheriton. 2016. EXCITE-VM: Extending the virtual memory system to support snapshot isolation transactions. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 401–412.

[27] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P Stevenson. 2014. SI-TM: reducing transactional memory abort rates through snapshot isolation. In *Proceedings of the 2014 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vol. 42. ACM, 383–398.

[28] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the 2017 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (ASPLOS '17)*. ACM, New York, NY, USA, 329–343. https://doi.org/10.1145/3037697.3037714

[29] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2014. Loose-ordering consistency for persistent memory. In *2014 32nd IEEE International Conference on*

Computer Design (ICCD). IEEE, 216–223.

[30] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys '17)*. ACM, New York, NY, USA, 499–512. https://doi.org/10.1145/3064176.3064215

[31] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *IEEE International Symposium on Workload Characterization (IISWC '08)*. Citeseer, 35–46.

[32] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems* 17, 1 (1992), 94–162. https://doi.org/10.1145/128765.128770

[33] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the 2017 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (ASPLOS '17)*. ACM, New York, NY, USA, 135–148. https://doi.org/10.1145/3037697.3037730

[34] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. 2002. Practical, transparent operating system support for superpages. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 89–104.

[35] Yuanjiang Ni, Jishen Zhao, Daniel Bittman, and Ethan L. Miller. 2018. Reducing NVM Writes with Optimized Shadow Paging. In *Proceedings of HotStorage '18*.

[36] Matheus Ogleari, Ethan L. Miller, and Jishen Zhao. 2018. Steal but no force: Efficient Hardware-driven Undo+Redo Logging for Persistent Memory Systems. In *Proceedings of the 24th International Symposium on High-Performance Computer Architecture (HPCA 2018)*. http://www.ssrc.ucsc.edu/ogleari-hpca18.pdf

[37] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSS: a full system simulator for multicore x86 CPUs. In *2011 48th Design Automation Conference (DAC)*. IEEE, 1050–1055.

[38] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceedings of the 41th Int'l Symposium on Computer Architecture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

[39] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. 2008. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development* 52, 4/5 (July 2008), 465–480. http://www.ssrc.ucsc.edu/PaperArchive/raoux-ibmjrd08.pdf

[40] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems* 10, 1 (Feb. 1992), 26–52. http://www.ssrc.ucsc.edu/PaperArchive/rosenblum-tocs92.pdf

[41] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters* 10, 1 (2011), 16–19.

[42] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, Todd C Mowry, and Trishul Chilimbi. 2015. Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. In *Proceedings of the 42th Int'l Symposium on Computer Architecture*. IEEE, 79–91.

[43] Nir Shavit and Dan Touitou. 1997. Software transactional memory. *Distributed Computing* 10, 2 (1997), 99–116.

[44] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. 2017. Proteus: A flexible and fast software supported hardware logging approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 178–190.

[45] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. 2008. The missing memristor found. *Nature* 453 (May 2008), 80–83. http://www.ssrc.ucsc.edu/PaperArchive/strukov-nature08.pdf

[46] Madhusudhan Talluri and Mark D. Hill. 1994. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '94)*. ACM, New York, NY, USA, 171–182. https://doi.org/10.1145/195473.195531

[47] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)*.

[48] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S Unsal. 2011. DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory. In *2011 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 340–349.

[49] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*. http://www.ssrc.ucsc.edu/PaperArchive/volos-asplos11.pdf

[50] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. 2014. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys '14)*. ACM, 26.

[51] Xiaojian Wu, Sheng Qiu, and A. L. Narasimha Reddy. 2013. SCMFS: A File System for Storage Class Memory and its Extensions. *ACM Transactions on Storage* 9, 3 (Aug. 2013). http://www.ssrc.ucsc.edu/PaperArchive/wu-tos13.pdf

[52] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*. http://www.ssrc.ucsc.edu/PaperArchive/xu-fast16.pdf

[53] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*. 167–181.

[54] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2540708.2540744